From (Meta) Objects to Aspects: A Java and AspectJ Point of View

Pierre Cointe[1], Hervé Albin-Amiot[1,2], and Simon Denier[1]

[1] OBASCO group, EMN-INRIA, LINA (CNRS FRE 2729),
École des Mines de Nantes, 4 rue Alfred Kastler, La Chantrerie,
44307 Nantes Cedex 3, France
{Pierre.Cointe, Herve.Albin-Amiot, Simon.Denier}@emn.fr
[2] Sodifrance, 4, rue du Château de l'Éraudière, 44324 Nantes, France

Abstract. eWe point some major contributions of the object-oriented approach in the field of separation of concerns and more particularly design-patterns and metaobject protocols. We discuss some limitations of objects focusing on program reusability and scalability. We sketch some intuitions behind the aspect-oriented programming (AOP) approach as a new attempt to deal with separation of concerns by managing scattered and tangled code. In fact AOP provides techniques to represent crosscutting program units such as display, persistency and transport services. Then AOP allows to weave these units with legacy application components to incrementally adapt them. We present a guided tour of `AspectJ` illustrating by examples the new concepts of pointcuts, advices and inter-type declarations. This tour is the opportunity to discuss how the `AspectJ` model answers some of the issues raised by post-object oriented programming but also to enforce the relationship between reflective and aspect-oriented languages.

1 Lessons from Object-Oriented Languages

More than twenty years of industrial practices have clearly enlightened the contributions but also the limitations of object-oriented technologies when dealing with software complexity [15]. Obviously, OO languages have contributed to significant improvements in the field of software engineering and open middleware [30,9]. Nevertheless, programming the network as advocated by `Java` made clear that the object model[33] even extended with the design of reusable micro-architectures such as patterns or frameworks was not enough to deal with critical issues such as scalability, reusability, adaptability and composability of software components [2,19].

In this introduction, we develop some limitations but also two of the main contributions of the OO. approach. These "pro and cons" put together, have challenged new open research ideas in the field of programming languages design and contributed to the emergence of new paradigms such as aspect-oriented programming [19,31].

F.S. de Boer et al. (Eds.): FMCO 2004, LNCS 3657, pp. 70–94, 2005.

1.1 Limitations (CONS)

A first source of problems when programming in the large, is the lack of mechanisms to modularize crosscutting concerns and then to minimize code tangling and code scattering[1]. A second source of problems is the difficulty of representing micro-architectures by using only classes and their methods. A third source of problems is the need of mechanisms to incrementally modify the structure or the behavior of a program. Considering object-oriented programming as THE final technololgy to solve these issues has made clear some well known drawbacks [9,10,15]:

1. Classes schizophrenia: as already quoted by Borning in 1986, classes play too many roles and there is some confusion around the concerns of a class as an object generator, a class as a method dispatcher and an (abstract) class as a part of the inheritance graph.
2. Granularity of behavioral factoring: when expressing behavioral concerns there is no intermediate level of granularity between a method and a class. For instance, there is no way in Java to factorize and then manipulate a set of methods as a whole. Similarly, a Java package - seen as a group of related classes - has not direct manipulable representation at the code level. Then, there is a real need for stateless groups of methods à la trait [28] to implement and compose mixin modules.
3. Class inheritance and transversal concerns: inheritance is not the solution for reusing crosscutting non-functional behaviors such as security or display that are by essence non hierarchical. For instance in `Java`, even very elementary state-less concerns such as being colorable, traceable, memoizable, movable, paintable, clonable, runnable, serializable, ... must be expressed by interfaces to be reused. Unfortunately, these interfaces do not provide any method implementations but only method specifications, limitating reusability.
4. Design patterns traceability: patterns provide reusable micro-architectures based on explicit collaborations between a group of classes [16]. Unfortunately they have no direct representation (reification) at the programming language level raising traceability and understandability issues [18].

1.2 Contributions (PRO)

On the one hand, the *Model View Controller* developed for `Smalltalk` has provided the user with a problem-oriented methodology based on the expression and the combination of (three) separate concerns related to user-interfaces design. The *MVC* was the precursor of *event programming* - in the Java sense - and contributed to make explicit the notion of *join point*, *e.g.,* some well defined points

[1] As stated in [31], *crosscutting* concerns refer to functionalities which do not naturally fit in usual module boundaries, *scattering* can be observed when a functionality must be called from many places and *tangling* when an individual operation may need to refer to many functionalities.

in the execution of a *model* used to dynamically weave the codes associated to the *view* and the *controller*.

On the other hand, object-oriented languages have demonstrated that *reflection* was a general conceptual framework to clearly modularize implementation concerns and to separate them from the functional/business logic. The principle is to introduce a metalevel description and two operations to switch between the base level (user) to this metalevel (implementor). Nevertheless, reflection is solution oriented since it relies on the protocols of the language to build a new solution by opening the system [29].

Our purpose is to develop now those two OO contributions to point later some interesting relationships between objects, design patterns and aspects.

The Model-View-Controller (MVC) was the first attempt to make the notion of concerns explicit when designing the user interface. *MVC* was also the inspirator of the well known `Observer` design pattern (see 3.3).

The main idea was to separate, at the design level, the *model* itself describing the application as a class hierarchy and two separate concerns: the *display*

```
public class Counter extends Object{
  private int value;
  public int getValue(){
    return value;
  }
  public void setValue(int nv){
    value=nv;
    // this.changed();
  }
  public int incr(int delta){
    this.setValue(value+delta);
    return delta;
  }
 public void incr(){
   this.incr(1);
  }
  public void raz(){
    this.setValue(0);
  }
  public String toString(){
    return "@" + value;
  }
  public static void main(String[] args) {
    Counter c1 = new Counter();
    c1.incr(); c1.incr(6); c1.raz();
  }
}
```

Fig. 1. The Counter class

and the *controller*, themselves described as two other separate class hierarchies. At the implementation level, standard object encapsulation and class inheritance were not able to express these crosscutting concerns and not able to provide the coupling between the model, its view, and its controller. This coupling necessitated:

- the introduction (for instance at the root level of the class model) of a *dependence mechanism* in charge of notifying the observers when a source-object state changes. This mechanism is required to automatically update the display when the state of the model changes.
- the instrumentation of (some) methods of the model to raise an event each time the state of the model changed, e.g, each time a given instance variable gets a new value.

To discuss in more details the *MVC* pattern, we transpose for `Java` the well known `Counter` example used by `Smalltalk` teachers. The principle is to develop the model first, as the `Counter` class, and then to introduce its associated `CounterView` and `CounterController` classes.

The Counter Class provides two accessors methods and some basic behaviors such as incrementing, reseting and "stringing" (representing as a text). None of these methods makes any assumption about the views and the controllers used to build the user-interface. Indeed, the associated `CounterView` and `CounterController` classes are defined separately.

Nevertheless, to use this `Counter` class according to the *MVC* paradigm, the developer has to manually manage state changes by inserting a `this.changed()` sentence every time the `value` field receives a new value. If the class is well designed, this insertion can be localized in only one point. In our case, since `incr`, `raz`, ... refer to it, only the setter method `setValue` has to be modified.

The CounterView Class aggregates its `Counter` but also its `CounterController`. The constructor establishes the dependant link between the `Counter` model and its view. In fact, every time the counter executes a `this.changed()`, the view will by notified by receiving an `update()` message. This update will refresh the view by redisplaying the new value of the `Counter` model.

Obviously the first challenge raised by the *MVC* was to automate the transformation of the model and the generation of the associated views and controllers. The second challenge was to proceed this generation in a non invasive way from the model side.

Metalevel Architectures à la Smalltalk and à la CLOS have clearly illustrated the potential of reflection to deal with separation of concerns[30]. The reflective approach makes the assumption that it is possible to separate in a given application, its *why* expressed at the base level, from its *how* expressed at the metalevel.

In the case of a reflective object-oriented language *à la Smalltalk*, the principle is to reify at the metalevel its structural representation, *e.g.,* its classes,

```
public class CounterView extends View {
  private Counter model, CounterController controller;
  private CounterView() {
    model = new Counter();
    model.addDependent(this); // dependency link
  }
  public void update(){
    model.getValue().toString().displayAt(...);
  }
}
public class CounterController extends MouseMenuController {
   private Counter model, CounterView view;
   ...
}
```

Fig. 2. The CounterView and MouseMenuController classes

their methods and the error-messages but also its computational behavior, *e.g.,* the message sending, the object allocation and the class inheritance. Depending on which part of the representation is accessed, reflection is said to be structural or behavioral. Meta-objects protocols (MOPs) are specific protocols describing at the meta-level the behavior of the reified entities. Specializing a given MOP by inheritance, is the standard way [8,17] to extend and to open the base language with new mechanisms such as explict metaclasses [3], multiple-inheritance, concurrency, distribution [24], aspects [4] or reified design patterns.

The design of metaobject protocol such as ObjVlisp, CLOS or ClassTalk contributed to the development of techniques to introspect and to intercess with program structures and behaviors [7,2,26,3]. The minimal *ObjVlisp* model was built upon two classes: `Object` the root of inheritance tree, `Class` the first metaclass and as such the root of the instantiation link, plus `MethodDescription`, the reification of object methods. Then object creation (structural reflection) and message sending (behavioral reflection) can be expressed as two compositions of primitive operations defined in one of these three classes[2]:

- Class.allocate 0 Object.initialize
- Class.lookup 0 MethodDescription.apply

In the case of an open middleware [23], the main usage of behavioral reflection is to control message sending by interposing a metaobject in charge of adding extra behaviors/services (such as transaction, caching, distribution) to its base object. Nevertheless, the introduction of such *interceptors/wrappers* metaobjects requires to instrument the base level with some *hooks* in charge of causally connecting the base object with its metaobject. Those metaobjects prefigured the introduction of AspectJ *crosscuts*, *e.g.,* the specification of execution points where extra actions should be woven in the base program [20,13].

[2] The dot notation `Class.allocate` meaning the `allocate` method defined in the `Class` class.

2 The Java Class Model and Its Associated MOP

The `Java` model is close to the ObjVlisp one, the main difference being that `Class` - the first metaclass - is **final** making `Class` and then the associated class model non extensible [7]. In fact, the Java reflective API[3] provides mainly a self-description of its class architecture and a related MOP mainly dedicated to its introspection.

2.1 Exposing the Java Class Model

On Figure 3, we recognize the `Object` class and the `Class` metaclass. Then the `Constructor`, `Field` and `Method` classes reify the three main kind of Java class members. Each of them implement the `Member` interface and specialize the `AccessibleObject` class introduced to interact with the security manager and get its authorization to intercede with the fields of an object. This simplified figure summarizes also - at the level of every class and using the common *Smalltalk* notation - the names of the metaobject protocols from which we would like to point out:

1. `Class.forName` to reify a class and then to get the reification of its different members,
2. `Class.newInstance` and `Constructor.newInstance` to allocate new objects,
3. `Method.invoke` to call compiled method,
4. `Field.get` and `Field.set` to read/write instance variables.

These methods give access to the description and then the control of some key events associated to OO program execution; respectively object creation, message sending and field references. They can be used as developed below to explicit and then to specialize by inheritance the associated mechanisms.

2.2 Using the Java MOP

ReflectiveObject. One key issue when using reflection is to control the execution flow by monitoring some key events (called hooks [29] or join points [20]) such as field accessing or message sending. The idea is to superimpose additional behaviors before, instead of, or after such events for instance to check pre or post conditions. In that perspective, the Java MOP can be used to explicit and then monitor the accesses to members (field or method) by introducing such hooks to execute extra code. This MOP makes possible the expression of the following rewriting rules expliciting method calls and field references in a *Smalltalk* way:

1. $o.selector(arg_2...arg_n) \hookrightarrow o.receive(\text{“}selector'', arg_2, ...arg_n)$
2. $o.field \hookrightarrow o.instVarAt(\text{“}field'', value)$
3. $o.field = value \hookrightarrow o.instVarAtPut(\text{“}field'', value)$

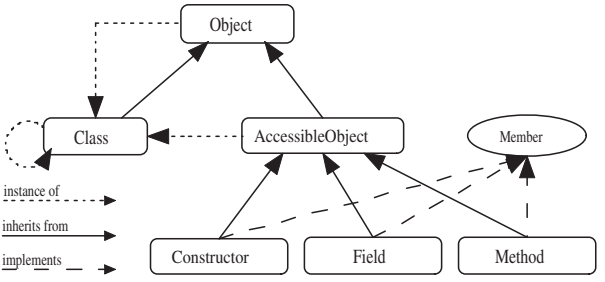
[3] We present here the `java.lang.reflect` package as provided by Java 1.4.

The `ReflectiveObject` class uses the Java MOP to explicit these transformation rules via its `receive` and `instVarAt` methods.

```
public class ReflectiveObject {
  public Object receive(String selector, Object[] args) {
    Method mth = null; Object r = null; Class[] classes = null;
    int lo = 0;
    if (args != null) {
      lo = Array.getLength(args);
      classes = new Class[lo];
     }
    for (int i = 0; i < lo; i++) {classes[i] = args[i].getClass();}
   try {
     // LOOKUP join point
     mth = getClass().getMethod(selector, classes);
    // before method call
     r = mth.invoke(this, args); // APPLY join point
    // after method call
          } catch (Exception e) {System.out.println(e); }
    return r;
  }
  public Object receive(String selector) {
    return receive(selector, null);
  }
  public Object receive(String selector, Object arg1) {
    return receive(selector, new Object[] { arg1 });
  }
  public Object receive(String selector, int arg1) {
    return receive(selector, new Object[] { new Integer(arg1) });
  }
  public void instVarAtPut(String name, Object value) {
    try {
    Field f = this.getClass().getDeclaredField(name);
    if (!Modifier.isStatic(f.getModifiers()))
      f.setAccessible(true);
    // before field reference
      f.set(this, value); // SET reference join point
    // after field reference
    } catch (Exception e) {
       System.out.println("ReflectiveObject.instVarAtPut "+ e);}
  }
}
```

More precisely:

1. `receive` : computes the signature of the associated message by extracting the class of every argument (selector and classes), looking up for an associated method in the class of the receiver (composition of `getClass` and `getMethod`)




```
java.lang.Object(getClass, clone)
    java.lang.Class(getName, newInstance, forName,
                    getDeclaredMethods, getDeclaredFields,
                    getDeclaredConstructors)
    java.lang.reflect.Member(getName, getDeclaringClass, getModifiers)
    java.lang.reflect.AccessibleObject(isAccessible, setAccessible)
       java.lang.Reflect.Field(get, set, setInt)
       java.lang.Reflect.Method(invoke)
       java.lang.Reflect.Constructor(newInstance)
```

Fig. 3. The Java class model and its associated MOP

and then apply this method to its arguments (`invoke`). Obvioulsy, `receive` can be specialized to call extra methods before, around or after the APPLY join point[4].

2. `instVarAtPut`: computes the representation of a field given by its name (composition of `getClass` and `getDeclaredField`) checks if it is an instance or a class variable (`isStatic`), turns the accessible security right to true (`setAccesible`) and change its value (`set`). Consequently, as soon as `Counter` is defined as a subclass of `ReflectiveObject`, its `Counter.setValue` method body can be rewritten as: `this.instVarAtPut (‘‘value’’, new Integer(nv))`. Then, `instVarAtPut` could be overridden to notify the dependents. More generally, this method can be specialized to call extra methods, before, after and around the SET reference join point.

MemoizingObject. The idea is to specialize the previous `receive` method to memoize the already computed results in a cache. For simplification purpose, we made the assumption that `receive` takes only one argument with type `int`. The implementation is as follows; when a `receive` is executed, we extract the first argument `n` and wrap it to a Java `Integer`, we check if the result as already been computed and cached for this argument. If true we directly returns the memoized result. If false, we call the super method in charge of realizing the computation and we record the result:

[4] `receive` allows also the computation of the selector to perform à la Smalltalk the method to call: `new ReflectiveObject().receive(‘‘get’’ + ‘‘class’’)`

```
public class MemoizingObject extends ReflectiveObject {
  public static HashMap cache = new HashMap();
  public Object receive(String selector, Object[] args) {
    int n = ((Integer)args[0]).intValue();
    Integer N = new Integer(n);
    if (cache.containsKey(N)){return (Integer)cache.get(N);}
    else {
      Integer r = (Integer)super.receive(selector, args);
      cache.put(N,r);
      return r;
    }
  }
}
```

The example below develops how to use this memoization concern to compute factorial numbers by the way of the `EFP.fact` method expressed in term of the `MemoizingObject.receive` method:

```
public class EFP extends MemoizingObject {
  static EFP f = new EFP();
  public static int fact(Integer n){
    int pn = n.intValue();
    if (pn == 0)
      return 1;
    else
      return pn * ((Integer)f.receive("fact",
                   (new Integer(pn - 1)))).intValue();
  }
  public static void main(String[] args) {
    try {
      System.out.println("The cache " + memo);
      Integer r= (Integer)f.receive("fact", 4);
      System.out.println("The cache " + memo + r);
      r= (Integer)f.receive("fact", 5);
      System.out.println("The cache " + memo + r);
        } catch (Exception e) {
          System.err.println("MemoizingObject.Main" + e);
        }
  }
}

/* Running EFP.main will produce :
The cache{}
The cache{2=2, 4=24, 1=1, 3=6, 0=1}24
The cache{2=2, 4=24, 1=1, 3=6, 5=120, 0=1}120
*/
```

ClassInspector uses the previous MOP to introspect a given `Java` class and pretty print its "signature". The main idea is to get its direct superclass, its

interfaces and then to enumerate the signature of its different declared members: fields, methods and constructors.

```
public class ClassInspector {
  private java.io.PrintStream out = System.out;

  public void inspect(Class aClass) {
    Field[] fields; Method[] methods; Constructor[] constructors;
    Class[] interfaces = aClass.getInterfaces();
    out.print(aClass.toString() + " extends "
        + aClass.getSuperclass().getName());
    for (int j = 0; j < interfaces.length; j++) {
      out.println("implements" + interfaces[j].getName());
    }
    out.println("");
    out.println("--> declared fields");
    fields = aClass.getDeclaredFields();
    for (int j = 0; j < fields.length; j++)
      out.println("   " + fields[j].toString());
    out.println("--> declared methods");
    methods = aClass.getDeclaredMethods();
    for (int j = 0; j < methods.length; j++)
      out.println("   " + methods[j].toString());
    out.println("--> declared constructors");
    constructors = aClass.getDeclaredConstructors();
    for (int j = 0; j < constructors.length; j++)
      out.println("   " + constructors[j].toString());
  }
  public static void main(String[] args) {
    ClassInspector desc = new ClassInspector();
    try {
      desc.inspect(Class.forName("fmco.Counter"));
    } catch (ClassNotFoundException e) {
      System.err.println(e);
    }
  }
}
```

Annex A.2 gives an example of using `ClassInspector` to "pretty print" our `Counter` class. The purpose is to check the members introduced by the `AspectJ` weaver, as soon as `Counter` is crosscutted by the `CounterObserver` aspect (see 3.3).

2.3 Some Drawbacks of the Java MOP

The previous examples have demonstrated how the `Java` API (via the `Reflective Object` class) can be used to incrementally modify the behavior of a program by controlling message sending or field accessing. Nevertheless the proposed solutions are difficult to generalize since:

- as for the `fact` and `setValue` examples, we need to manually transform regular `Java` codes to explicit the usage of `receive` and `instVarAtPut`,
- contrary to `Smalltalk`, the associated rewriting rules have to deal with `Java` primitive types and necessitates the introduction of casts and wrappers which make the translations more complex,
- `EFP` and `Counter` have to subclass `ReflectiveObject` to get the associated reflective behaviors. Since `Java` only provides single inheritance it is a strong constraint for a class to use this specialization mechanism to get reflective facilities.

At the conceptual level, we can reformulate these drawbacks as the difficulty to systematically reify message sending and field accessing in a non-invasive way and as the problem of modularizing extra code performed "around" such events without using standard class inheritance. At the implementation level, we should discuss the extra execution cost induced by metaprogramming, and the inherent complexity of opening the Java class model [29].

3 A Guided Tour of AspectJ

"A characteristic of aspect-oriented programming, as embodied in AspectJ, is the use of advice to incrementally modify the behavior of a program. An advice declaration specifies an action to be taken whenever some condition arises during the execution of the program. The condition is specified by a formula called a pointcut designator. The events during execution at which advice may be triggered are called joint points. In this model of aspect-oriented programming, join points are dynamic in that they refer to events during the execution of the program [32]."

`AspectJ` is a general purpose language built as a super set of `Java` (see [1] and chaper 6 of [14]). The main idea is to introduce a new unit called an aspect in charge of modularizing crosscutted concerns. This unit looks like a class definition but supports the declaration of pointcuts and advice. These pointcuts are used by a specific compiler to weave the advice with regular `Java` code.

From an industrial perspective, it is the first largely diffused language used to develop or reengineer relevant applications according to aspect-oriented design [14]. From an academic perspective, `AspectJ` is historically the first aspect-oriented language and the natural candidate to expose the relationships between objects, metaobjects and aspects by answering some issues raised by post-object-oriented programming.

3.1 The Join Point and Advice Models

The main intuition behind AOP is to introduce a *join point* model raising events every time an interesting point is reached during the execution of a program. Then the idea is to propose a *pointcut language* to select specific join points and an an *advice language* to express some extra code to be woven at those pointcuts. In the case of `AspectJ` both the pointcut language and the advice language are extensions of `Java`. More precisely and revisiting [19] we propose the following definitions:

- Join point: a well defined point in the execution of a program. As an extension of Java, `AspectJ` proposes about ten different kinds of those points related to object-oriented execution; method call, method execution, field reference (get and set), constructor call, (static) initializer execution, constructor execution, object (pre) initialization, exception handler execution [21].
- Pointcut (*when*): an expression designating a set of join points that optionally exposes some of the values in the associated execution context. These pointcuts can be either user-defined or primitives. These pointcuts can be composed (like predicates) according to three logical operators : logical and (&& operator), logical or (|| operator) and logical negation (! operator).
- Advice (*how/what*): a declaration of what an aspect computes at intercepted join points. In fact a method like mechanism used to declare that certain code should execute when a given pointcut matched. The associated code can be told to run before the actual method starts runing, after the actual method body has run and instead/around the actual method body. Notice that `AspectJ` provides a reification of the current join point by introducting the new `thisJoinPoint` pseudo variable (see the `DummyTrace2` aspect in 3.2).
- Inter-type declaration (introductions): declarations of members that cut across multiple classes or declarations of change in the inheritance relationship between classes. In a reflective way, those declarations are used to open a class by statically introducing new members or by changing its super class or super interfaces.
- Aspect: a modular unit of crosscutting implementation, composed of pointcuts and advice, plus ordinary `Java` member declarations. An `AspectJ` aspect declaration has a form similar to that of a `Java` class declaration.

The rest of this section is a guided tour of `AspectJ` introducing step by step the previous concepts and illustrating them with examples. We will distinguish between behavioral crosscutting affecting the run time behavior and static crosscutting affecting the class and object structures (page 185 of [21]).

3.2 Behavioral Crosscutting

The principle is to modularize simple crosscutting concerns such as monitoring, tracing and memoizing with `AspectJ` aspects and then to adapt existing `Java` classes such as `Counter` and `FP` by weaving their associated advice.

Introducing Pointcuts (Counter and the Daemon aspect). Coming back to the `Counter` class exposed in Figure 1, we still want to notify a dependent (an observer) every time its `value` field is changed. Contrary to the MOP approach, we expect to proceed without editing the definition of the `Counter` class and without evading its `setValue` method. The `AspectJ` solution is to modify automatically and *a posteriori* the code of the `Counter` class by weaving the advice associated to the `Daemon` aspect. This aspect allows to monitor the write access to the `Counter.value` field as follows:

- declaration of a user defined `changed` poincut designator associated to the `Counter.value` set event:
- declaration of an `after` advice which method body will be executed after a set event occur. More precisely after a logical combination of a `changed` **and** an `args` **and** a `target`. The two primitives pointcuts (`args` and `target`) are used to catch - in the dynamic context - the values of the current receiver `r` and the current argument `n`.

```
public aspect Daemon {
  // pointcut designator
  pointcut changed(): set(int Counter.value);
  // after advice
  after(int n, Counter r): changed() && args(n) && target(r){
    r.getDependent().update(n);
  }
}
```

Obviously, poincuts enables to abstract over control flow [25]. In particular, it becomes easy to materialize the execution trace of different method calls. The next three examples develop how to define trace aspects within `AspectJ`.

Introducing Before and After Advice (DummyTrace1). The idea is to define a `DummyTrace` aspect to visualize the computation of the two mutually recursive methods `FP.odd` and `FP.even` are associated to the `FP` (standing for Functional Programming) class. defined in Annex 5.1. The first expected trace looks like the left part of Figure 4. We define the `DummyTrace1` aspect as follows:

- a `traceCounter` field counting the number of the traced method calls. It is used to indent the outputs,
- two `calleven` and `callodd` poincut descriptors monitoring the `FP.even` and `FP.odd` method calls,
- four advice in charge of wrapping these method calls by printing in the standard output the value of the `n` argument **before** the associated pointcut and the value of the same argument plus the computed `b` result `after` the pointcut.

```
-->Before calleven(3)
 -->Before callodd(2)
  -->Before calleven(1)
   -->Before callodd(0)
   false<--After callodd(0)
  false<--After calleven(1)
 false<--After callodd(2)
false<--After calleven(3)
```

```
-->call(FP.even(..)) 3
 -->call(FP.odd(..)) 2
  -->call(FP.even(..)) 1
   -->call(FP.odd(..)) 0
   false<--call(FP.odd(..)) 0
  false<--call(FP.even(..)) 1
 false<--call(FP.odd(..)) 2
false<--call(FP.even(..)) 3
```

Fig. 4. FP class crosscuted by the DummyTrace aspects (DummyTrace1 at left, DummyTrace2 at right)

```
public aspect DummyTrace1 {
  int traceCounter=0;
  void indent(){
    for (int j = 0; j < traceCounter; j++) {System.out.print(" ");}
  }

  pointcut calleven(): call(static boolean FP.even(int)) ;
  pointcut callodd(): call(static boolean FP.odd(int)) ;

  before(int n): calleven() && args(n){
    traceCounter++; indent();
    System.out.println("-->Before calleven(" + n + ")");
  }
  after(int n) returning (boolean b) : calleven()&& args(n){
    indent();
    System.out.println(b + "<--After calleven(" + n + ")");
    traceCounter--;
  }
  after(int n) returning (boolean b) : callodd()&& args(n){
    indent();
    System.out.println(b + "<--After callodd(" + n + ")");
    traceCounter--;
  }
  before(int n): callodd() && args(n){
    traceCounter++; indent();
    System.out.println("-->Before callod(" + n + ")");
  }
}
```

Using Wildcards in Pointcut Signature (DummyTrace2). Our first version of the trace aspect can be improved by i) using only one pointcut designator to declare the methods to be traced and then by ii) factorizing the two before/after advice.

- `AspectJ` supports the use of wild cards in the signatures of pointcuts. The `*` character matches any number of characters and `..` matches zero and more arguments [21]. Consequently the expression `static boolean FP.*(int)` designates all the static methods defined in `FP` taking only one `int` as argument and returning a `boolean`.
- `AspectJ` introduces also the `thisJoinPoint` pseudo-variable to provides reflective dynamic information about the kind of join point, its signature and its context.

By combining wildcard and join point reification, we get a more concise and more generic aspect which corresponding execution trace is given in the right part of Figure 4.

```
public aspect DummyTrace2 {
  int traceCounter=0;
  void indent(){
    for (int j = 0; j < traceCounter; j++) {System.out.print(" ");}
  }
  pointcut boolean_FP_int(): call(static boolean FP.*(int)) ;

  before(int n): boolean_FP_int() && args(n){
    traceCounter++; indent();
    System.out.println("-->" + thisJoinPoint.toShortString() +  " " + n);
  }
  after(int n) returning (boolean b) : boolean_FP_int()&& args(n){
    indent();
    System.out.println(b + "<--" + thisJoinPoint.toShortString() + " " + n);
    traceCounter--;
  }
}
```

Introducing Abstract Aspect (the TraceProtocol Aspect). Since its pointcut descriptor explicitly refers to the `FP` class, the `DummyTrace2` aspect cannot be reused to trace any kind of method in any kind of class. But quoting page 340 of [20]): *"AspectJ provides a simple mechanism of pointcut overriding and advice inheritance. To use this mechanism a programmer defines an abstract aspect, with one or more abstract pointcuts, and with advice on the poincut(s). This, then, is a kind of library that can be parameterized by aspects that extend it".*

To make the trace aspect reusable, we introduce the **abstract** `TraceProtocol` aspect, its associated **abstract** `trace` pointcut descriptor and its two before/after advice:

```
abstract aspect TraceProtocol {
  int traceCounter=0;
  void indent(){
    for (int j = 0; j < traceCounter; j++) {System.out.print(" ");}
  }

  abstract pointcut trace();

  before(int n): trace() && args(n){
    traceCounter++;
    indent();
    System.out.println("-->" + thisJoinPoint.toShortString() +  " " + n);
  }
  after(int n) returning (int r): trace() && args(n){
    indent();
    System.out.println(r + "<--" + thisJoinPoint.toShortString() + " " + n);
    traceCounter--;
  }
}
```

Then we can provide different concrete implementations of this trace protocol. For instance, to trace the `fact` and `fib` static methods also associated to `FP` together with the `incr` method of `Counter` we define the `Tracing` aspect as an extension of `TraceProtocol`:

```
public aspect Tracing extends TraceProtocol {
  pointcut trace() :
    call(int Counter.incr(int))  ||
    call(static int FP.fact(int))  ||
    call(static int FP.fib(int));
}
```

Introducing Around Advice (the Memoizing Aspect). The `around` advice allows to use a join point for adding extra behavior to the proceeding of the standard code execution or by replacing this standard code execution by a totally new code. The `Memoizing` aspect illustrates how to use it in the case of the `FP.fact` static method. As for `MemoizingObject` class in section 2.2, the principle is to memoize all the already computed results in a `cache`. Then when a new call occurs, if the result is already cached then it is directly returned, otherwise the computation is done in a regular way due to the **proceed** construction, the result cached and returned:

```
public aspect Memoizing {
  public static HashMap cache = new HashMap();

  pointcut callfact(): call(static int FP.fact(int)) ;

  int around(int n): callfact() && args(n){
    Integer N = new Integer(n);
    if (cache.containsKey(N)){
      return ((Integer)cache.get(N)).intValue();
    }
    else {
      int r = proceed(n);
      cache.put(N, new Integer(r));
      return r;
    }
  }
}
```

A few enhancements would transform this skeleton aspect into a more generic, multiple methods caching aspect. Compared to the usage of reflection and the definition of the `MemoizingObject` we observe a symetry between the `super` and `proceed` as two constructs allowing to call some overridden behaviors. Obviously the `AspectJ` solution looks better since the definition of the `FP` class is not impacted by the definition of a `Memoizing` concern.

Composing Memoizing and Tracing. `AspectJ` automatically compose different crosscutting aspects. In the case of `FP`, after the definition of the `Memoizing` and `Tracing` aspects and their associated `callfact` and `trace` pointcuts, every call to the `FP.fact` static method will be memoized **and** traced. As shown by Figure 5, after a first call of `FP.fact(3)` (left part) the results of fact(0), fact(1), fact(2) and fact(3) are cached, then when calling `FP.fact(5)`, only the computations of `FP.fact(5)` and `FP.fact(4)` are proceeded (right part):

```
fact(3)
 -->call(FP.fact(..)) 3
  -->call(FP.fact(..)) 2
   -->call(FP.fact(..)) 1
    -->call(FP.fact(..)) 0
    1<--call(FP.fact(..)) 0
   1<--call(FP.fact(..)) 1
  2<--call(FP.fact(..)) 2
 6<--call(FP.fact(..)) 3
fact(3)=6
```

```
fact(5)
 -->call(FP.fact(..)) 5
   -->call(FP.fact(..)) 4
    -->call(FP.fact(..)) 3
    6<--call(FP.fact(..)) 3
   24<--call(FP.fact(..)) 4
 120<--call(FP.fact(..)) 5
fact(5)=120
```

Fig. 5. FP crosscutted by the Memoizing and Tracing aspects

Threading: The Concurrent Aspect. One recurrent question is how to make concurrent the execution of objects. `Java` suggests to use a `Runnable` interface in charge of adapting a class to the `Java` concurrency model. An `AspectJ` alternative programming idiom is to replace the standard execution of a `main` static method by the launching of a new instance of an anonymous `Thread`. Coming back to the clock example discussed in [5] we get:

```
public aspect Concurrent {
  void around() : execution(public static void Clock.main(String[])) {
    new Thread(){
      public void run() {
        System.out.println("Started in another thread");
        proceed();
      }
    }.start();
  }
}
```

3.3 Structural Crosscutting

In the tradition of reflective architecture, `AspectJ` provides a mechanism known as *inter-types declaration* to open `Java` classes (and interfaces) by introducing new members or by changing the inheritance relationship between classes (and interfaces). Quoting the "Introduction to `AspectJ`" available at [1]; *"unlike advice, which operates primarily dynamically, introduction operates statically, at compile-time."*

`AspectJ` supports four kinds of such declarations introductions: field, method and constructor introductions plus class hierarchy alteration. Their syntax is as follows:

1. `modifiers type ClassName.newFieldName [= expression]`
   adds the `newFieldName` to the `ClassName` class with optionally the initial value associated to `expression`,
2. `modifiers type ClassName.newMethodName(parameters) body`
   adds the `newMethodName` to the `ClassName` class with the corresponding `parameters` and method `body`,

3. `modifiers type InterfaceName.addedMethodName(parameters) body` adds the `newMethodName` to **all** the classes implementing the `InterfaceName` class with the corresponding `parameters` and method `body`,
4. `declare parents: ClassName extends SuperClassName` `SuperClassName` becomes the new direct superclass of `ClassName`,
5. `declare parents: ClassName implements ListOfInterfaceNames` `ClassName` implements the new set of `ListOfInterfaceNames`.

In this section, we address the issue of representing the `Observer` design pattern as an aspect. We revisit the presentation of [18] about implementing in `AspectJ` the different design patterns presented in [16]. Obviously, this is an opportunity to reintroduce the `Counter` example and to come back to the `Smalltalk` *MVC*.

The Observer Design Pattern as an Aspect. Quoting [16, page 294], the intent of the `Observer` pattern is to *"define a one-to-many dependency between objects so that when one object changes state, all its dependents are notified and updated automatically"*. The key roles in the `Observer` design patten are `subject` and `observer`. Here we made the assumption of the existence of two interfaces, respectively `Subject` and `Observer`, plus a `Printer` class, all of them defined in Annex A.2. Then the idea is to use a `CounterObserver` aspect to adapt the `Counter` and `Printer` classes to play the roles of `subject` and `observer`.

As shown by Figure 6, the aspect has to adapt the `Counter` class to make it implement the `Subject` interface and in the same time to adapt the `Printer` class to make it implement the `Observer`.

Notice that [12] uses "interface" introductions to address another issue discussed in 1.1 of providing an implementation of traits (set of stateless related methods) in `Java` [28].

The PatternObserverProtocol Aspect. modularizes the update logic and the registration logic for observers. The update logic is handled by the after `stateChanges` advice in charge of updating the list of observers whereas the registration logic is due to a set of introductions. Seven of them are in charge of adding the members `subject`, `getSubject()`, `setSubject()`, `observers`, `addObserver()`, `removeObserver()`, `getObservers()`, to the `Observer` and `Subject` classes. Notice finally how the abstract aspect is parameterized both by the abstract `stateChanges(Subject)` pointcut and the `update` abstract method.

The CounterObserver Aspect. specializes the `PatternObserverProtocol` for a configuration where `Counter` plays the role of a `Subject` and `Printer` the role of an `Observer`. The introduction mechanism is used for this configuration task. First we declare that `Counter` implements `Subject` and `Printer` implements `Observer`. Then we define a concrete `update` method for the `Printer` class. Finally we declare the `stateChanged` pointcut to target and monitor every call of the `Counter.setValue(int)` method.

```
aspect CounterObserver extends PatternOberverProtocol {
  pointcut stateChanges(Subject s):
    target(s) && call(public void Counter.setValue(int));

  declare parents: Counter implements Subject;
  declare parents: Printer implements Observer;

  public void Printer.update() {
    this.print("update occured in Counter" + this.getSubject());
   }
  }
```

Testing the DemoPatternObserver. An important property of this design is the lack of coupling between `Counter`, `Printer` and the `PatternObserverProtocol` abstract aspect. In fact, all coupling are localized in `CounterObserver` and some client such as `Client.main` on Figure 8. Obviously, another aspect can specialize `PatternObserverProtocol` to define its own schema involving other classes. It is one possible reusable implementation of the Observer design pattern: however it is not a universal one. For example, only one instance of the pattern/aspect is allowed per subject, so there is no distinction between non-related observers when updating.

Further Reading About the Join Point Model. This presentation of `AspectJ` is quite short and does not present all the details of its join point model. In particular we do not say anything about so called *scoping join point* matching any join point where the associated code is defined within a given scope or based on the control flow in which they occur. For more details see [1,21].

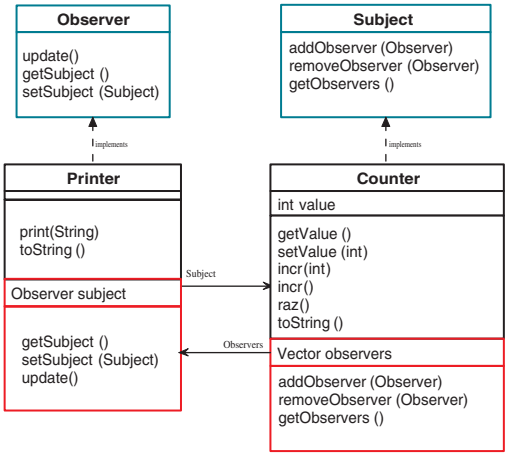


Fig. 6. Counter as a Subject in the Observer design pattern

```
abstract aspect PatternObserverProtocol {
  abstract pointcut stateChanges(Subject s);
  // Update Logic
  after(Subject s, int value): stateChanges(s) && args(value){
    for (int i = 0; i < s.getObservers().size(); i++) {
      ((Observer)s.getObservers().elementAt(i)).update();
    }
  }
  // Registration Logic
  private Vector Subject.observers = new Vector();
  public void Subject.addObserver(Observer obs) {
    observers.addElement(obs);
    obs.setSubject(this);
  }
  public void  Subject.removeObserver(Observer obs) {
    observers.removeElement(obs);
    obs.setSubject(null);
  }
  public Vector Subject.getObservers() { return observers; }
  private Subject Observer.subject = null;
  public void Observer.setSubject(Subject s) { subject = s; }
  public Subject Observer.getSubject() { return subject; }
}
```

Fig. 7. The Observer Design Pattern as an Aspect

```
public static void Client. main(String[] args) {
   Counter c1 = new Counter();
   Printer scribe = new Printer();
   c1.raz();
   scribe.print(c1.toString());
   c1.addObserver(scribe);
   c1.incr(3);
   scribe.print(c1.toString());
   c1.raz();
   scribe.print(c1.toString());
}
/* Will produce
[Printer] @0
 -->call(Counter.incr(..)) 3
[Printer] update occured in Counter@3
 3<--call(Counter.incr(..)) 3
[Printer] @3
[Printer]update occured in Counter@0
[Printer] @0
*/
```

Fig. 8. Counter crosscutted by the CounterObserver and Tracing aspects

4 Conclusion and Open Questions

In this paper we have discussed reflective and aspect-oriented languages two fields of research boosted by the object-oriented community. For pointing out the continuum between objects, metaobjects and aspects we have chosen `Java` and `AspectJ` as the two test beds to express crosscutting concerns in a modular way. Nevertheless, as demonstrated by the proceedings of conferences such as Reflection, ICFP or AOSD (see `http://aosd.net`), meta-entities and aspects are not limited to OO languages but can impact the design of all programming languages including functional, logical and constraint based ones.

In this area, promising open research issues includes providing operational and formal semantics for advice and pointcut models [13,32], using the same general purpose language for defining advices and joint points versus developing multi-paradigm languages, exploring the application of domain specific languages to the definition of aspects, building a reflective kernel as a basis to implement aspect-oriented languages [5,27] and in the opposite direction using AOP as a basis for reflective and metalevel architectures [22], understanding the contribution of aspects to the field of generative programming [6,25], experimenting with the refactoring of legacy codes based on reusable aspects [18].

Acknowledgments

This work is part of the new AOSD network of excellence and its language laboratory (see `http://www.aosd-europe.net`). It benefited from previous discussions presented in [10,11].

References

1. AspectJ site. : See http://eclipse.org/aspectj
2. Aksit, M., Black, A., Cardelli, L., Cointe. P., Guerraoui, R. (editor), and al.: Strategic Research Directions in Object-Oriented Programming, ACM Computing Surveys, volume 8, number 4, page 691-700, (1996).
3. Bouraqadi-Sadani , M.N. , Ledoux, T., Rivard F.: Safe Metaclass Programming. Proceedings of OOPSLA 1998. Editor Craig Chambers,ACM-Sigplan, pages 84-96, volume 33, number 10, Vancouver, British Columbia, USA, October 1998.
4. Bouraqadi-Sadani , M.N. , Ledoux, T.: Supporting AOP Using Reflection. Chapter 12 of [14], pages 261-282, 2005.
5. Chiba, Shigeru.: Generative Programming from a Post Object-Oriented Programming ViewPoint. Proceedings of the Unconventional Programming Paradigms workshop. To appear as LNCS volume. Mont St Michel, France, 2005.
6. Czarnecki, K., Eisenecker, U.W.: Generative Programming. Methods, Tools, and Applications. Addison-Wesley (2000).
7. Cointe, P.: Metaclasses are First Class: The ObjVlisp Model. Proceedings of the second ACM SIGPLAN conference on Object-oriented Programming, Systems, Languages, and Applications (OOPSLA 1987). Editor Jerry L. Archibald, ACM SIGPLAN Notices, pages 156-167, volume 22, number 12, Orlando, Florida, USA, October 1987.

8. Cointe, P.: CLOS and Smalltalk : a Comparison. Chapter 9, pages 215-250 of [26]. The MIT Press, 1993.
9. Cointe, P.: Les langages à objets. Technique et Science Informatiques (TSI), volume 19, number 1-2-3, pages 139-146, 2000.
10. Cointe, P., Noyé, J., Douence, R., Ledoux, T., Menaud, J.M., Muller, G., Sudholt, M.: Programmation post-objets. Des langages d'aspect aux langages de composants. RSTI série L'objet. volume 10, number 4, pages 119-143, 2004. See also http://www.lip6.fr/colloque-JFP.
11. Cointe, P.: Towards Generative Programming. Unconventional Programming Paradigms workshop, UPP 04. LNCS 3566, pp 302-312. J.-P. Banâtre et al. Editors. Springer Verlag, 2005.
12. Denier, S.: Traits Programming with AspectJ. RSTI série L'objet. Special issue on Aspect-Oriented Programming (to appear). See also pages 62-78 of the unformal proceeding at `http://www.emn.fr/x-info/obasco/events/jfdlpa04/actes/`, 2005.
13. Douence, R., Motelet, O., Sudholt, M.: A formal definition of crosscuts. Proceedings of the 3rd International Conference on Reflection 2001, LNCS volume 2192, pages 170-186, (2001).
14. Filman, E. R., Elrad, T., Clarke, S., Aksit, M.: Aspect-Oriented Software Development. Addison-Wesley, 2005.
15. Gabriel, R.: Objects Have Failed. See http://www/dreamsongs/com/Essays.html and also http://www.lip6.fr/colloque-JFP/
16. Gamma, E., Helm, R., Johnson. R., Vlissides, J.: Design Patterns. Elements of Reusable Object-Oriented Software. Addison-Wesley Professional Computing Series. 1995
17. Kiczales, G., Ashley, J., Rodriguez, L., Vahdat, A., Bobrow, D.: Metaobject Protocols Why We Want Them and What Else They Can Do. Chapter 4, pages 101-118 of [26]. The MIT Press, 1993.
18. Hannemann , J., Kiczales, G.: Design Pattern Implementation in Java and AspectJ. Object-Oriented Programming, Systems, Languages, and Applications. OOPSLA 2002. ACM SIGPLAN Notices, volume 37, number 11, pages 161-173.
19. Kiczales, G., Lamping, J., Mendhekar, A., Maeda, C., Lopes, C, Loingtier, J.-M., Irwin, J.: Aspect-Oriented Programming. ECOOP 1997 - Object-Oriented Programming - 11th European Conference, volume 1241, pages 220–242. 1997
20. Kiczales, G., Hilsdale, E., Hugunin, J., Kersten, M., Palm, J., Griswold, W.: An Overview of AspectJ ECOOP 2001 - Object-Oriented Programming - 15th European Conference, LNCS volume 2072, pages 327–354. 2001
21. Kiselev, I.: Aspect-Oriented Progamming with AspectJ. Sams Publishing, 2003.
22. Kojarski, S., Lorenz, D., Hirschfeld, R.: Reflective Mechanism in AOP Languages. Draft paper.
23. Ledoux, T.: OpenCorba: A Reflective Open Broker. Proceedings of the second international conference on Meta-Level Architectures and Reflection. Cointe, P.: editor. LNCS 1616, Pages 197-214, Saint-Malo, France, 1999.
24. McAffer, J,: Meta-level Programming with CodA. Proceedings of ECOOP 95. Page 190-214, Springer LNCS, Aarhus, Danemark, 1995
25. Mezini, M., Ostermann, K.: A Comparison of Program Generation with Aspect-Oriented Programming. Proceedings of the Unconventional Programming Paradigms workshop. To appear as LNCS volume. Mont St Michel, France, 2005.
26. Pæpcke, A.: Object-Oriented Programming : The CLOS perspective. The MIT Press, 1993.

27. Rodriguez, L., Tanter, E., Noyé J,: Supporting Dynamic Crosscutting with Partial Behavioral Reflection : a Case Study. RSTI série L'objet. Special issue on Aspect-Oriented Programming (to appear). See also pages 118-137 of the unformal proceeding at `http://www.emn.fr/x-info/obasco/events/jfdlpa04/actes/`, 2005.
28. Scharli, N., Ducasse, S., Nierstrasz, O., Black, P.: Traits: Composable Units of Behaviour. ECOOP 2003 - Object-Oriented Programming - 17th European Conference, Editor L. Cardelli. LNCS volume 2743, pages 248–274. 2003
29. Tanter, E., Noyé, J., Caromel, D., Cointe, P.: Partial Behavioral Reflection: Spatial and Temporal Selection of Reification. Proceedings of the 18th ACM SIGPLAN conference on Object-Oriented Programming, Systems, Languages, and Applications. OOPSLA 2003. ACM SIGPLAN Notices, volume 38, number 11, pages 27-46.
30. Thomas, D.: Reflective Software Engineering - From MOPS to AOSD. Journal Of Object Technology, volume 1, number 4, pages 17-26. October 2002.
31. Wand, M.: Understanding Aspects. Invited talk at ICFP 2003. Available at www.ccs.neu.edu/home/wand/ICFP 2003.
32. Wand, M., Kiczales, G., Dutchyn, C.: A semantics for Advice and Dynamic Join Points in AOP. ACM Toplas, volume 26, issue 5, pages 890-910. 2004.
33. Wegner, P.: Dimensions of Object-Based Language Design. Proceedings of the second ACM SIGPLAN conference on Object-oriented programming, systems, languages, and applications (OOPSLA 1987). Editor Jerry L. Archibald, ACM SIGPLAN Notices, pages 168-182, volume 22, number 12, Orlando, Florida, USA, October 1987.

A Annex

A.1 FP: Our Testbed (Ugly Class)

```
public class FP extends ReflectiveObject{
  public static int fact(int n) {
    if (n == 0)
      return 1;
    else
      return n * fact(n - 1);
  }
  public static int fib(int n) {
    if (n <= 1)
      return 1;
    else
      return fib(n - 1) + fib(n - 2);
  }
   public static boolean even(int n) {
     if (n == 0)
       return true;
     else
       return odd(n - 1);
  }
   public static boolean odd(int n) {
     if (n == 0)
       return false;
```

```
class fmco.Counter extends java.lang.Object implements fmco.Subject
--> declared fields
   private int fmco.Counter.value
   public java.util.Vector fmco.Counter.ajc$...$observers
--> declared methods
   public static void fmco.Counter.main(java.lang.String[])
   public java.lang.String fmco.Counter.toString()
   public int fmco.Counter.getValue()
   public void fmco.Counter.setValue(int)
   public void fmco.Counter.incr()
   public void fmco.Counter.incr(int)
   public void fmco.Counter.raz()
   public void fmco.Counter.addObserver(fmco.Observer)
   public java.util.Vector fmco.Counter.getObservers()
   public void fmco.Counter.removeObserver(fmco.Observer)
--> declared constructors
   public fmco.Counter()
```

Fig. 9. Counter crosscutted by CounterObserver

```
    else
      return even(n - 1);
  }
  public static void main(String[] args) {
    System.out.println("fact(5)=" + fact(5));
    System.out.println("fib(4)=" + fib(4));
  }
}
```

A.2 The Printer Class and the Subject and Observer Interfaces

```
public interface Subject {
  public void addObserver(Observer o );
  public void removeObserver(Observer o);
  public java.util.Vector getObservers();
}
public interface Observer {
  void setSubject(Subject s);
  Subject getSubject();
  void update();
}
public class Printer {
  public void print(String s){
    System.out.println("[Printer] " + s);
  }
  public String toString(){
    return "aPrinter";
```

```
  }
  public static void main(String[] args) {
    new Printer().print("Hello Word");
  }
}
```

A.3 Inspecting Counter Crosscutted by CounterObserver

The reader can check the presence of the `obervers` field as the one of the `addObserver`, `getObservers`, `removeObserver` methods, all of them introduced by the `CounterObserver` aspect.